Layered connectors: revisiting the formal basis of architectural connection for complex distributed systems

Conference or Workshop Item

Layered Connectors

Revisiting the Formal Basis of Architectural Connection for Complex Distributed Systems

Amel Bennaceur[1]⋆ and Valérie Issarny[2]

[1] The Open University, Milton Keynes, UK
[2] Inria Paris-Rocquencourt, France

Abstract. The complex distributed systems of nowadays require the dynamic composition of multiple components, which are autonomous and so complex that they can be considered as systems in themselves. These components often use different application protocols and are implemented on top of heterogeneous middleware, which hamper their successful interaction. The explicit and rigorous description and analysis of components interaction is essential in order to enable the dynamic composition of these components. In this paper, we propose a formal approach to represent and reason about interactions between components using *layered connectors*. Layered connectors describe components interaction at both the application and middleware layers and make explicit the role of middleware in the realisation of this interaction. We provide formal semantics of layered connectors and present an approach for the synthesis of layered connectors in order to enable the dynamic composition of highly heterogeneous components. We validate our approach through a case study in the area of collaborative emergency management.

Keywords: Component interaction, Layered connectors, Middleware, Dynamic composition, Architectural mismatches

1 Introduction

In 1994, Allen and Garlan published their seminal paper on formalising architectural connection [1], for which they received the ICSE most influential paper award 10 years later. The authors put forward a vision, and a supporting theory, that improved our understanding of software architecture by relying on the elegance of formal methods to highlight the relation between the different entities of a software system. These entities are *components*, which are meant to encapsulate computation, and *connectors*, which are meant to encapsulate interaction [22].

At the same time, another vision that focuses on the implementation of distributed systems has received an increasing attention among developers, that of *middleware*. Middleware is a software entity logically placed between the application and the operating system that provides an abstraction that facilitates the communication and coordination of distributed components [25]. Fortunately,

⋆ This work was performed when the author was at Inria.

the two visions are by no means antagonistic. Indeed, the influence of middleware on the architecture of software systems has long been recognised [21] and it has been admitted that middleware plays an important role in implementing connectors [15, 19]. However, this influence has not been explicitly formalised and the relation between connectors and middleware remains ill defined. In this paper, we show how the formalisms used in the literature to describe and analyse architectural connection can be extended to reason about components interaction at the middleware layer. Considering both the software architecture and the middleware perspectives allows us to better understand the digital world surrounding us and also empowers us with methodologies to solve many of the problems inherent to this complex digital world.

One critical problem is that of *architectural mismatches* [9]. Architectural mismatches occur when composing two, or more, software components to form a system and those components make conflicting assumptions about their environment. Components may exhibit disparate data types and operations, and may have distinct business logics, which results in *application heterogeneity*. Components may also rely on different communication standards (e.g., CORBA or SOAP) which define disparate data representation formats and induce different architectural constraints, which results in *middleware heterogeneity*. Architectural mismatches must be solved in order to enable components to be composed successfully. Since connectors model the exchange of information between components and the coordination of their behaviours, solving architectural mismatches often amounts to finding or creating the appropriate connector that enables their successful interaction. This connector acts as a translator that performs the data conversions necessary to solve differences between components' interfaces and as a controller that coordinates components' behaviours. The implementation of this connector should also consider the different middleware solutions used by the components involved.

As the modern digital world become increasingly populated with mobile and ubiquitous computing technology, the scope and boundary of software systems can be uncertain and can change. As a result, the connectors that regulate components interaction cannot be designed and implemented beforehand, but rather synthesised dynamically. Although much work has been carried out on connector synthesis [13], existing solutions have not fully succeeded in keeping pace with the increasing complexity and heterogeneity of modern software, and meeting the demands of runtime support. Solutions either (i) focus on application heterogeneity and generate the connector that enable the composition of the components, based on some domain knowledge, but fail to deploy them on top of heterogeneous middleware [18, 24, 12, 26, 17], or (ii) deal with middleware heterogeneity while assuming developers to provide all the data translations and behavioural coordinations that need to be made, as is the case with Enterprise Service Buses (ESB) [10]. At the best of our knowledge, only Starlink [6] attempts to tackle both application and middleware heterogeneity by providing a runtime execution engine that allows developers to deploy translators and controllers dynamically. However, it is the role of the developers to specify these translators and controllers, which might be somehow restrictive considering the domain expertise necessary to provide these specifications.

We argue that architectural mismatches are a cross-cutting concern and solutions thereof must consider both application and middleware heterogeneity. On the one hand, the application layer provides the appropriate level of abstraction to reason about architectural mismatches and synthesise the appropriate connectors based on knowledge specific to the application domain. On the other hand, the middleware layer offers the necessary services for realising the synthesised connector and instantiating the specific data structures and protocols expected by the components at hand. Therefore, we propose a rigorous approach to model and reason about components interaction from the application down to the middleware layer. The objective is to provide a systematic solution for solving architectural mismatches. To this end, we make the following contributions:

- *Formalisation of components interactions at both the application and middleware layers.* We build upon pioneering work on the formalisation of architectural connection by Allen and Garlan [1] to describe the role of middleware in the formal description of connectors. The goal is to identify the mechanisms used by middleware solutions to coordinate the behaviours of components and their influence on components interaction regardless of the specific middleware implementation. We also make explicit the semantics of actions used by the components, using ontologies. The result is the formal definition of *layered connectors* that explicitly describe the coordination and the data exchange between components at both the application and middleware layers. Consequently, we can verify the ability to specify and implement connectors regulating the interaction between highly heterogeneous components.
- *Synthesis of layered connectors in order to solve architectural mismatches.* We define an approach that exploits recent advances in both the fields of software engineering and distributed systems to enable the synthesis of layered connectors in order to allow the composition of heterogeneous components. Note that rather than focusing on a specific technique for translator or controller synthesis, which we tackle elsewhere [4], we show how these techniques can be made to work together in order to solve application and middleware heterogeneity.
- *Experimentation with a real-world scenario.* To validate our approach, we consider one representative application domain, that of emergency management, as illustrated by the GMES[3] initiative. GMES gives a special interest to the support of emergency situations (e.g., forest fire) across different European countries. Indeed, each country defines an emergency management system that encompasses multiple components that are autonomous, designed and implemented independently, and do not obey any central control or administration. Nonetheless, there are incentives for these components to be composed and collaborate in emergency situations. In [2], we used this scenario to illustrates the role of models@runtime is supporting interoperability; in this paper, we specifically focus on the formal specification and synthesis of layered connectors to allow the dynamic composition of heterogeneous components.

The paper is structured as follows. Section 2 describes background work. Section 3 presents the formal semantics of layered connectors and presents our

[3] Global Monitoring for Environment and Security –`http://www.gmes.info/`

approach for their synthesis. Section 4 illustrate the approach using the emergency management scenario. Finally, Section 5 concludes the paper and discusses future work.

2 Background on Connectors

In this section we introduce the foundational concepts of our approach and explain the relation with existing solutions for the formal description, synthesis, and implementation of connectors.

Formal Basis of Architectural Connection. We consider as our starting point the formalisation of architectural connection introduced by Allen and Garlan [1], which uses process algebra to model the behaviours of components together with their interaction. More specifically, we use FSP (Finite State Processes) [16] based on the follow-up work by Spitznagel and Garlan [24]. The behaviour of a component is modelled using *ports* while a connector is modelled as a set of *roles* and a *glue*. The roles specify the expected behaviours of the interacting components while the glue describes how the behaviours of these components are coordinated. The ports, roles, and glue are specified as FSP processes. The syntax of FSP is summarised in Table 1 while we will assume that the reader has some familiarity with FSP in what follows.

Definitions	
set S	Defines a set of action labels
$[i : S]$	Binds the variable i to a value from S
Primitive Processes (P)	
$a \rightarrow P$	Action prefix
$a \rightarrow P \vert b \rightarrow P$	Choice
$P(X =' a)$	Parameterised process: P is described using parameter X and modelled for a particular parameter value, $P(a1)$
$P/\{new\_1/old\_1, ...\}$	Relabelling
Composite Processes ($\|P$)	
$P \| Q$	Parallel composition
forall $[i : 1..n]$ $P(i)$	Replicator construct: equivalent to the parallel composition $P(1) \| ... \| P(n)$
$a : P$	Process labelling

Table 1. FSP syntax overview

A component can be attached to a connector only if its port is *behaviourally compatible* with the connector role it is bound to. Behavioural compatibility between a component port and a connector role is based upon the notion of refinement, which implies the inclusion of the traces of the expected behaviour of the component in those of the observed behaviour of the component [1]. In other words, it should be possible to substitute the role process by the port process. Verifying behavioural compatibility allows us to check the presence or absence of architectural mismatches. To solve architectural mismatches, we must find or create a connector whose roles are behavioural compatible with components' ports.

Synthesis of Connectors. It is not always possible to find an existing connector for managing interactions between heterogeneous components and it is difficult and time consuming to design and implement a new connector from scratch [19]. There are several compositional approaches for connector construction by reusing

existing connector instances [24]. Nevertheless, with the increasing emphasis on mobility and ubiquity of software systems, there is a growing interest on synthesis of connectors. Rather than expecting a developer to specify how the connector instances should be composed, solutions for connector synthesis seek to analyse the ports of components in order to generate the connector that enables their successful interaction. More specifically, the roles of this connector are assumed to be same as the ports of the components involved, and a glue is synthesised which guarantees that the components interact without errors (e.g., deadlocks) and exchange meaningful data.

Formal methods focus on the behaviour of components, which they rigorously analyse in order to reveal potential inconsistencies, ambiguities, and incompleteness. Once potential execution errors are detected, they can be solved either by eliminating the interactions leading to the errors or by introducing a controller that forces the components to coordinate their behaviours correctly. Only the introduction of a controller can keep the functionality of the system intact by enabling its components to achieve their individual functionalities. Existing solutions for the generation of controllers (e.g., [26, 17, 12]) often operate on a high-level abstraction, which makes turning the generated controller into an implementation challenging. Moreover, they often assume that the behaviours of the components are described using the same set of actions or the correspondence between the actions of components' interfaces is provided.

Semantic Web technologies allow us to infer the translations necessary to reconcile the differences between components' interfaces. Ontologies play a key role in the Semantic Web by formally representing shared knowledge about a domain of discourse as a set of concepts, and the relationships between these concepts [11]. Ontologies have been extensively used to automate the reasoning about the information exchanged between software components, especially in ubiquitous computing environments, so as to infer the translations necessary to reconcile the differences in the syntax of this information [18]. However, ontology reasoning techniques focus on differences at the application layer alone, assuming the use of the same middleware underneath.

Middleware to Implement Connectors. The implementation of a connector is often based on middleware since middleware provides reusable solutions that facilitate communication and coordination between components [15, 19]. However, while components and connectors are conceptually separate, middleware solutions are often invasive since they influence the implementation of the components as well. As a result, components implemented using different middleware solutions are not able to work together. For example, a SOAP client cannot invoke a REST service even if they use the same application data and obey the same business logic. Therefore, other middleware solutions have been proposed in order to reconcile the differences between middleware [10]. However, when these middleware solutions follow different interaction patterns, e.g., shared memory and publish/subscribe, the differences are such that they cannot always be solved [7].

The connector classification introduced by Mehta *et al.* [20] provides a convenient framework that helps selecting the appropriate connectors according to

application requirements. It is also used to create a set of guidelines that specify the conditions under which connectors can be composed. However, this set of guidelines are based on some intuitive understanding and rules of thumb and lack the formal basis necessary to make the solution sound and future proof.

The Need for Layered Connectors. In order to enable the dynamic composition of components, it is important to find the right level of abstraction so as to reason about the interaction of these components automatically while keeping enough details to turn the conclusions drawn during the reasoning phase into a concrete artefact. It is difficult to deal with implementation-level differences, as it involves managing many details that, although crucial, make the reasoning very difficult, if not impossible. But an excessive abstraction is also useless as the decision space toward refining the result of the reasoning and turning it toward a concrete solution would be immense. Furthermore, knowledge about the domain in which the components evolve is necessary in order to capture the meaning of the information they exchange.

We introduce the concept of layered connector in order to capture the application-level semantics of components interaction as well as the semantics of the associated middleware solution. Through the concept of layered connectors, we consolidate the techniques and solutions proposed in the fields of software engineering and middleware in order to describe the semantics of components interaction precisely. The goal is to reason about components interaction at a level of abstraction that would allow us to solve architectural mismatches by synthesising the appropriate layered connectors that act as (i) translators by ensuring the meaningful exchange of data between components, (ii) controllers by coordinating the behaviours of the components to ensure the absence of errors in their interaction, and (iii) middleware by enabling the interaction of components across the network so that each component receives the data it expects at the right moment and in the right format.

3 Formal Specification and Synthesis of Layered Connectors

We first show how the semantics of middleware solutions can be formalised using a combination of formal methods and ontologies. Then, we describe how to represent the relation between these middleware solutions and the application implemented on top. Finally, we describe how to synthesis layered connectors in order to enable heterogeneous components to interact successfully.

3.1 Middleware-layer Connectors

Communication in distributed systems is always based on low-level message passing as offered by the underlying network. Expressing communication through message passing is harder than using primitives proposed by middleware solutions [25]. While middleware solutions and implementations define diverse IDLs and message formats, their interaction protocols follow comparably few interaction patterns, a.k.a., communication paradigms/types [25] or coordination models/paradigms [10]. An interaction pattern defines the rules to coordinate the behaviours of the components. In Mehta *et al.* connector classification [20], these

interaction patterns match with the connector types that provide communication and coordination services. Our approach seeks to identify, capture and separate the core of a middleware solution, represented by the interaction pattern it uses, from specific details related to the format of messages. To this end, we introduce, for each interaction pattern, an ontology that models the essential primitives of this interaction pattern, which we use to specify the behaviours expected by the components implemented using a middleware solution based on this interaction pattern as well as how these behaviours are coordinated. A specific middleware solution is modelled using specialisation over the ontology that represents the interaction pattern on which the middleware solution is based. While in [14] we gave initial thoughts about an ontology for middleware solutions, the lack of behaviour description for the interaction patterns made it impossible to make a formal analysis of these solutions as well as to verify transformations between different interaction patterns.

Remote Procedure Call. Remote procedure call (RPC) [5] represents the most common interaction pattern in distributed systems. RPC directly and elegantly supports client/server interactions with servers offering a set of operations through a service interface and clients calling these operations directly as if they were available locally. The interaction is supported by a pairwise exchange of messages from the client to the server and then from the server back to the client, with the first message containing the operation to be executed at the server and associated arguments and the second message containing any result of the operation. To interact according to RPC, the client and the server must agree on the format of the messages they exchange as well as the encoding of the data, which represent the arguments and results, enclosed in these messages.

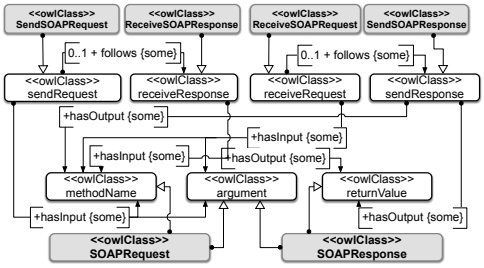


Fig. 1. The RPC ontology specialised with SOAP [14]

$$Client\ (X =' op) = (sendRequest[X] \rightarrow receiveResponse[X] \rightarrow Client).$$

$$Server\ (X =' op) = (receiveRequest[X] \rightarrow sendResponse[X] \rightarrow Server).$$

$$RPCGlue\ (X =' op) = (sendRequest[X] \rightarrow receiveRequest[X] \rightarrow sendResponse[X] \rightarrow receiveResponse[X] \rightarrow RPCGlue).$$

$$\|RPCInteraction = (\mathsf{forall}[op : Interface]\, Client(op)) \,\|\, (\mathsf{forall}[op : Interface]\, RPCGlue(op)) \,\|\, (\mathsf{forall}[op : Interface]\, Server(op))).$$

Fig. 2. RPC behavioural description

Figure 1 depicts the RPC ontology. The invocation of an operation is achieved using sendRequest, which specifies the operation invoked using methodName and the associated argument, possibly followed by a receiveResponse, which includes the operation invoked together with the results returnValue. The server gets the operation call using the receiveRequest primitive. If the result of this operation is not empty, the server returns it using the sendResponse primitive. Figure 1 further shows how the RPC ontology is specialised to describe SOAP. Note that even though SOAP supports the sending and reception of messages independently, it is often used to realise RPC-based interactions, especially in the context of

Web Services. In this context, SOAPRequest includes methodName and argument while SOAPResponse encompasses methodName and returnValue.

Figure 2 describes how the behaviours of the client and server are coordinated. The variable *op* defines the operation signature that is made up of the methodName, argument, and returnValue. The set of all operations signatures is denoted by *Interface*. The precise definition of the *Interface* set is specific to the application.

Distributed Shared Memory. Distributed Shared Memory (DSM) provides developers with a familiar abstraction of reading or writing (shared) data structures as if they were in their own local address spaces. A DSM-based middleware enables components to read and write data in the shared memory, regardless of the exact location of the data. Nevertheless, the structure of the shared data is defined at the application layer and the middleware does not provide any guarantee about when data is made available and how long it will reside in the shared memory. In other words, the synchronisation between the readers and writers also needs to be managed at the application layer.

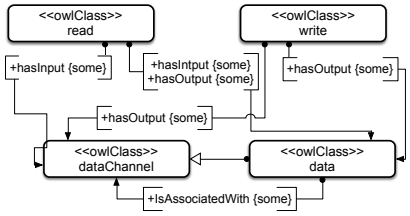


Fig. 3. DSM ontology [14]

$$
\begin{aligned}
&Writer(X =' data) = (write[X] \rightarrow Writer).\\
&Reader(X =' data, Y =' dataChannel) = (read[X][Y] \rightarrow Reader).\\
&SharedMemory(X =' data) = (write[X] \rightarrow P[X]),\\
&P[X][a : DataChannels] = (\mathsf{if}\ (X\ matches\ a)\\
&\qquad\qquad \mathsf{then}\ \ read[X][a : DataChannels]\\
&\qquad\qquad \rightarrow P[X]).\\
&\\
&\|DSMInteraction\ \ =(\ (\mathsf{forall}[data : Data]\, Writer(data))\\
&\qquad\qquad \|\ \ (\mathsf{forall}[data : Data]\, SharedMemory(data))\\
&\qquad\qquad \|\ \ (\mathsf{forall}[data : Data][dataChannel :\\
&\qquad\qquad DataChannels]\, Reader(data, dataChannel))).
\end{aligned}
$$

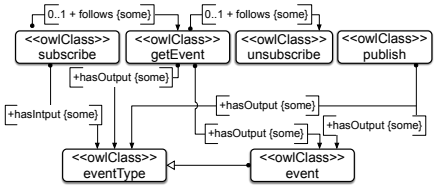
Fig. 4. DSM behavioural description

Figure 3 illustrates the DSM ontology. Two primitives are used: write, which adds data to the shared memory and read, which retrieves data from the shared memory. The dataChannel concept allows the selection of the data to read, while every data is associated with some dataChannel.

The coordination of the behaviours of components, which can be considered as readers or writers, is achieved through the shared memory as depicted in Figure 4. Since FSP supports only finite state models, we must represent *data* and *dataChannel* as sets. The precise definition of these sets depends on the application that uses the DSM. Note that there is one process P per data item, which deals with the several reads assuming that the data are persistent, i.e. the data can be read infinitely often. The *matches* function indicates whether the data channel specified in the read corresponds to the data managed by P. It is the role of the middleware to implement the *matches* function.

Publish/Subscribe. Many applications require the dissemination of information or items of interest from a large number of producers to a similarly large number of consumers. Publish/subscribe middleware solutions provide an intermediary service, a *broker*, that ensures that information generated by producers is delivered to the consumers that want to receive it. In other words, publish/subscribe middleware solutions (sometimes also called distributed event-based middleware) allow subscribers to register their interest in an event, or a pattern of events, and ensure that they are asynchronously notified of events

generated by publishers. The task of the publish/subscribe middleware is to match subscriptions against published events and ensure the correct delivery of event notifications. The expressiveness of publish/subscribe middleware solutions is determined by the type of event subscriptions they support: either subscriptions are made using specific topics (also referred to as subjects) which the events belong to, or based on the content of the event.



$$
\begin{aligned}
&Publisher(X =' event) = (publish[X] \rightarrow Publisher).\\
&Subscriber(X =' event, Y =' eventType)\\
&\qquad =(subscribe[Y] \rightarrow getEvent[Y] \rightarrow Subscriber).\\
&Broker \quad =P,\\
&P \quad =(\ subscribe[eventType : EventTypes]\\
&\qquad \rightarrow MATCH[eventType]\\
&\qquad \mid publish[Events] \rightarrow P),\\
&MATCH[eventType : EventTypes]\\
&\qquad =(\ publish[event : Events] \rightarrow\\
&\qquad \text{if } (event \text{ matches } eventType) \text{ then}\\
&\qquad\quad getEvent[event] \rightarrow MATCH[eventType]\\
&\qquad \text{else } MATCH[eventType]).\\
&\|PubSubInteraction=(\ (\text{forall}[event : Events]Publisher(event))\\
&\qquad \| \quad (Events : Broker)\{publish/Events.publish\}\\
&\qquad \| \quad (\text{forall}[event : Events][eventType :\\
&\qquad EventTypes]Subscriber(event, eventType))).
\end{aligned}
$$

Fig. 5. Publish/Subscribe ontology [14] **Fig. 6.** Publish/Subscribe behavioural description

Figure 5 depicts the Publish/Subscribe ontology. The subscribe primitive, which is parameterised by eventType that defines a filter over the set of all possible events, is used to express an interest in a set of events. The events are delivered to subscribers using getEvent. The unsubscribe primitive is used to revoke a subscription. The publish primitive is used to disseminate an event event to interested subscribers.

The behaviours of publishers and subscribers are coordinated using a broker as described in Figure 6. Similarly to DSM, we represent *event* and *eventType* as sets while the precise definition of these sets depends on the application that uses the publish/subscribe middleware. Note that we define several $MATCH$ processes, each of which manages the subscriptions related to one specific event type. The *matches* function indicates whether the published event is of the type managed by the specific $MATCH$ process. The middleware is in charge of implementing this function.

To sum up, there are different interaction patterns that define specific rules to coordinate the behaviours of components. While we present and formalise the interactions patterns most commonly used in the development of middleware solutions, we are aware that some middleware are not represented, e.g., stream-based middleware solutions. The case of streaming solutions is to be explored in future work.

3.2 Bridging the Application and Middleware Layers

Whether expressed as operation calls, data read and write, or event publication, component interactions mainly consists in the production and consumption of information. The production of information in the environment is modelled using provided actions while the consumption from the environment is modelled using required actions, with the understanding that required actions are received from and controlled by the environment, whereas provided actions are emitted and

controlled by the component. More specifically, a required action $\langle op, i, o \rangle$, where the symbols op, i, and o are references to concepts in a domain ontology $\mathcal{O}$, represents a consumption of a functionality op by sending the appropriate input data i and receiving the corresponding output data o. The dual provided action[4] $\langle \overline{op}, i, o \rangle$ uses the inputs and produces the corresponding output.

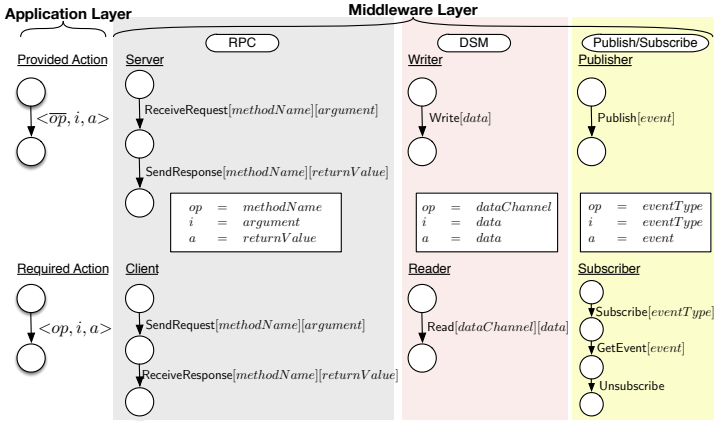


Fig. 7. Mapping interaction patterns primitives to required/provided actions

All middleware solutions, regardless of the interaction pattern they are based on, provide an abstraction that represents required and provided actions. Figure 7, which revisits that in [14], shows how the primitives associated with each interaction pattern, and defined in the associated ontology, are mapped to required/provided actions. In RPC, the server provides an action whose functionality is expressed by the *methodName*,it uses as input *argument* and generates *returnValue*. The associated client requires this same action. In DSM, it is the writer that provides an action while the functionality is enclosed in the data itself as *data* is associated with a specific *dataChannel*. The reader selects *data* available on some *dataChannel*.In publish/subscribe, the publisher provides an action whose functionality is represented by the event type. The subscriber selectively consumes these events by subscribing to a specific *eventType*, recalling that each *event* is associated with some *eventType*.

The formalisation of middleware interaction patterns allows us to define, and verify, transformations between required actions implemented using one interaction pattern and provided actions implemented using another interaction pattern. Furthermore, since every middleware solution specialises some interaction pattern, these transformations can also be specialised with the primitives of specific middleware solutions. For example, consider the case of a required action implemented using RPC and a provided action implemented using DSM, i.e. interaction between Writer and Client[5]. We can specify a transformation between Writer and Client that consists in intercepting the request and converting the *methodName* and *arguments* into *dataChannel*. Then, using *dataChannel* to read

[4] We use the overline as a convenient shorthand to denote provided actions
[5] The description of all other possible cases can be found in [3].

data, which is transformed into the appropriate *returnValue* and sent back as a response to the client. This is formally specified as follows:

$$
\begin{array}{ll}
Client\ (X =' op) & = (sendRequest[X] \rightarrow receiveResponse[X] \rightarrow Client).\\
Writer(Y =' data) & = (write[Y] \rightarrow Writer).\\
RPC2DSMGlue(X =' op, Y =' dataChannel, Z =' data) & = (receiveRequest[X] \rightarrow translate[X][Y]\\
 & \quad \rightarrow read[Y][Z] \rightarrow translate[Z][X] \rightarrow sendResponse[X] \rightarrow RPC2DSMGlue).\\
\|RPC\text{-}DSM & = (\ (\mathsf{forall}[op : Interface] Client(op))\ \|\ (\mathsf{forall}[op : Interface] RPCGlue(op))\\
 & \quad \|\ \ (\mathsf{forall}[data : Data] Writer(data))\ \|\ (\mathsf{forall}[data : Data] SharedMemory(data))\\
 & \quad \|\ \ (\mathsf{forall}[op : Interface][data : Data][dataChannel : DataChannels]\\
 & \qquad RPC2DSMGlue(op, data, dataChannel))).
\end{array}
$$

where the sets *Interface*, *Data*, and *DataChannels*, as well as the translations $translate[X][Y]$ and $translate[Z][X]$ performed by *RPC2DSMGlue*, are specific to the application. We can easily verify that $\|RPC\text{-}DSM$ is free from deadlocks. Note that the specification of this connector depends on the translations performed at the application layer. In the subsequent section, we show how these transformations between interaction patterns can help implementing the layered connector that regulates components interaction from the application down to the middleware layer.

3.3 Synthesis of Layered Connectors

To enable the dynamic composition of highly-heterogeneous components, i.e. components featuring differences at both the application and middleware layers, we must synthesise the layered connector that ensures that each component receives the data it expects at the right moment and in the right format. Because of space considerations and because the focus of the paper is on describing an approach to solve architectural mismatches between highly-heterogeneous components rather than on devising a specific approach for translator or controller synthesis, we will present the gist of each synthesis step while details can be found elsewhere [3].

The first step consists in using domain knowledge, which is represented using the adequate domain ontology, to calculate the correspondences between the actions required by one component and those provided by the other, that is *translator synthesis* (see Figure 8, ❶). For each correspondence identified, we associate a *matching process* that synchronises with each component and performs the translations necessary to reconcile the differences in the syntax of the input/output data used by each component.

The second step consists in composing the matching processes in a way that guarantees that the components will reach their termination states without errors such as deadlocks, that is *controller synthesis* (see Figure 8, ❷). In [3] we propose an approach that combines constraint programming and ontology reasoning to compute the correspondences between the actions used by the components, which we then use to synthesise the controller.

Finally, *concretisation* entails the instantiation of the data structures expected by each component and their delivery according to the interaction pattern defined by the middleware based on which the component is implemented (see Figure 8, ❸). To this end, we rely on the mappings defined in Section 3.2 to refine the matching processes. In addition, the middleware ontologies, which are specialised with the middleware solutions used by each component (see Figure 1), serve specialising the transformations between the different interaction patterns. We also assume that *parsers and composers* dedicated to specific middleware solutions, can be

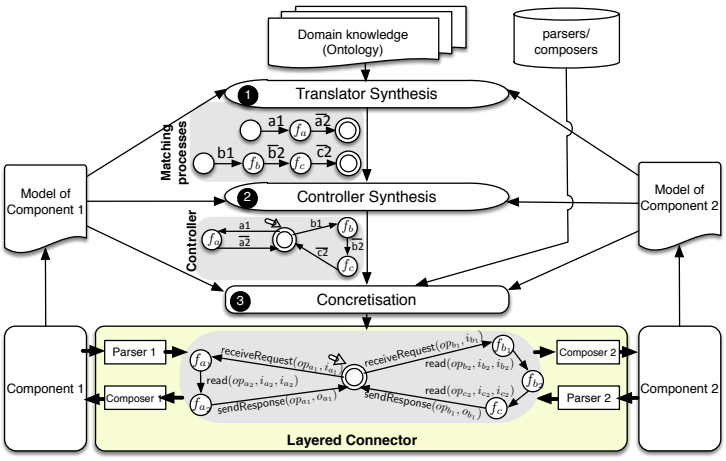


Fig. 8. Overview of our approach to the synthesis of layered connectors

used. A middleware-specific parser intercepts network messages conforming to the associated middleware solution and processes them in order to extract the relevant data. For example, a SOAP parser allows us to access the methodName and argument fields without a need to parse the network messages. In a dual manner, a middleware specific composer creates adequate network messages given the necessary data. For example, a SOAP composer allow us to create an appropriate SOAP response by simply giving the methodName and returnValue. More specifically, we rely on the Starlink framework [6] to generate parsers and composers for different middleware solutions.

4 Layered Connectors in Action: The GMES Case

To provide insight into the benefits of using the synthesis of layered connectors to support the dynamic composition of heterogeneous components, we now present the experiment we conducted in the context of the GMES initiative [8]. GMES is the European Programme for the establishment of a European capacity for Earth Observation. In particular, a special interest is given to the support of emergency situations (e.g., forest fire) across different European countries. GMES makes a strong case of the need for solutions to enable multiple, and most likely heterogeneous, components to collaborate in order to perform the different tasks necessary for decision making. These tasks include collecting weather information, locating the agents involved, and monitoring the environment.

Figure 9 depicts the case where the emergency system of *Country 1* is composed of a Command and Control centre (*C2*) which takes the necessary decisions for managing the crisis based on the information about the weather provided by the *Weather Service* component, the positions of the various agents in field given by *Positioning-A*, and the video of the operating environment captured by *UGV* (Unmanned Ground Vehicle). The components of *Country 1* use SOAP to communicate. *Country 2* assists *Country 1* by supplying components that provide *C2* with extra information. These components are *Weather Station*, *Positioning-B*, and *UAV* (Unmanned Aerial Vehicle). However, *C2* cannot use

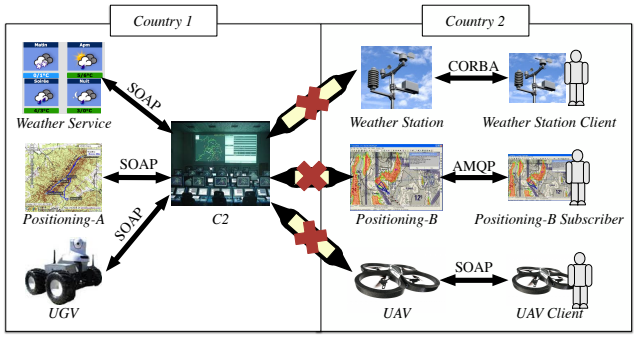


Fig. 9. The GMES example

these components directly. Indeed, *Weather Station* is implemented using CORBA and provides specific information such as temperature or humidity whereas *Weather Service*, which is used by *C2*, returns all of this information using a single operation. Furthermore, *Positioning-A* is implemented using SOAP whereas *Positioning-B* is implemented using AMQP and hence communicates according to publish/subscribe. Furthermore, *UGV* requires the client to login, then it can move in the four cardinal directions while *UAV* is required to takeoff prior to any operation and to land before logging out. Table 2 summarises the differences between *Country 1* and *Country 2* components. We refer the interested reader to [8] for further details about each component. To enable *C2* to use the components provided by *Country 2*, the appropriate layered connectors have to be synthesised. For space considerations we only describe the Weather case in the following; the detailed description of all the cases can be found in [3].

Case	Application Differences	Middleware Differences
Weather	one-to-many	SOAP vs. CORBA
Positioning	one-to-one	SOAP vs. AMQP (RPC vs. Pub/Sub)
Vehicle Control	extra actions	—

Table 2. Application and middleware differences in GMES cases

The interface of *C2* includes three required actions *login*, $getWeather$, and *logout*. *C2* first logs in, invokes $getWeather$ several times, and finally logs out. Since *C2* interacts using SOAP, then each of the required actions is realised by invoking the appropriate operation *op*, which belongs to the set $\{login, getWeather, logout\}$, by sending a SOAP request and receiving a SOAP response, which is formalised as follows:

$$\begin{aligned}
&\textsf{set}\ C2_weather_actions = \{login, getWeather, logout\}\\
&C2_weather_role \quad = (req.login \rightarrow P1),\\
&P1 \quad = (req.getWeather \rightarrow P1 \mid req.logout \rightarrow C2_weather_role).\\
&SOAPClient\ (X =' op) = (req.[X] \rightarrow sendSOAPRequest[X] \rightarrow receiveSOAPResponse[X]\\
&\qquad \rightarrow SOAPClient).
\end{aligned}$$

The interface of *Weather Station* encompasses three provided actions *login* $getTemperature$, $getHumidity$, and *logout*. *Weather Station* expects clients to login first, then ask for the temperature or humidity several times, and log out to terminate. Note that the two actions $getTemperature$ and $getHumidity$ can be performed independently. For each provided action, *Weather Station* receives a CORBA request, which it processes, and then sends the corresponding response:

$$
\begin{array}{ll}
\textsf{set}\ WeatherStation\_actions = \{login, getTemperature, getHumidity, logout\} & \\
WeatherStation\_role & = (prov.login \rightarrow P2), \\
P2 & = (\ prov.getTemperature \rightarrow P2 \mid prov.getHumidity \rightarrow P2 \\
 & \quad \mid\ \ prov.logout \rightarrow WeatherStation\_role). \\
CORBAServer\ (X =' op) & = (prov.[X] \rightarrow receiveCORBARequest[X] \\
 & \quad \rightarrow sendCORBAResponse[X] \rightarrow CORBAServer).
\end{array}
$$

The first step is to compute the necessary translations between the actions of *C2* and *Weather Station* given some knowledge about the application domain represented by the GMES ontology [8]. Beside the semantic correspondences between the *login* and *logout* required and provided actions, there is also one between the *getWeather* action required by *C2* and the sequence of actions *getTemperature* and *getHumidity* provided by *Weather Station*. Once the correspondence identified, we must compute the associated translation functions. Therefore, in addition to the domain ontology, we also use XML schema matching techniques to identify related elements between the schema of the input/output data of the actions [23].

Each correspondence is associated with a matching process. Note though that *getWeather* may be translated into *getTemperature* followed by *getHumidity* or *getHumidity* followed by *getTemperature*, which results in some ambiguity with which the controller must deal by selecting one of the matching processes. This selection may be motivated by some non-functional property or the length of the sequences of actions. In our example, let us assume that the selected matching process translates the *getWeather* action required by *C2* into the sequence of *getTemperature* followed by *getHumidity* provided by *Weather Station*. In addition, the controller must compose the matching processes in the right order, i.e. first matching the *login* actions, then *getWeather* with *getTemperature* followed by *getHumidity*, and finally the *logout* actions. The resulting controller is as follows:

$$
\begin{array}{ll}
Controller & = (req.login \rightarrow prov.login \rightarrow P), \\
P & = (req.getWeather \rightarrow prov.getTemperature \rightarrow prov.getHumidity \rightarrow P \\
 & \quad \mid\ req.logout \rightarrow prov.logout \rightarrow Mediator).
\end{array}
$$

Finally, the concretisation step involves dealing with differences between the middleware solutions used to implement the two components. Let *SOAPImpl* and *CORBAImpl* denote the middleware-layer connectors associated with the SOAP and CORBA middleware solutions respectively, each of which is associated with dedicated parsers and composers. Even though the format of the requests/responses is different, the interaction pattern is the same and can be transformed into primitives from the RPC ontology. The resulting layered connector ($\|\, WeatherSystem$) is described as follows:

$$
\begin{array}{l}
\|\, WeatherSystem = (C2\_weather\_role \ \|\ WeatherService\_role \ \|\ Controller \\
\quad \|\ (\textsf{forall}[op : C2\_weather\_actions]\ SOAPImpl(op)) \\
\quad\ /\{sendSOAPRequest/sendRequest, receiveSOAPResponse/receiveResponse\} \\
\quad \|\ (\textsf{forall}[op : WeatherStation\_actions]\ CORBAImpl(op)) \\
\quad\ /\{receiveCORBARequest/receiveRequest, sendCORBAResponse/sendResponse\}).
\end{array}
$$

We can verify (using LTSA) that the synthesised layered connector is free from deadlocks.

To evaluate the performance of our approach, we measured the time necessary to execute each step of the synthesis. The results are reported in Table 3. While the controller synthesis, which is performed using the approach described in [3] and involves FSP behavioural analysis, takes few milliseconds to execute, the translator synthesis and the concretisation necessitates around $1s$ as they also requires dealing with XML and ontology processing. Still, the overall time for the

synthesis of layered connectors remains less than $2s$. Furthermore, the synthesis is performed only once and is definitely faster than hand-coding the layered connector or even specifying it. In summary, the synthesis of layered connectors allows us to deal with architectural mismatches by reconciling the differences in the implementations of components at both the application and middleware layers.

Case	Weather	Positioning	Vehicle Control
Translator Synthesis	10031	9709	10256
Controller Synthesis	2	<1	7
Concretisation	809	903	465

Table 3. Processing time (in ms) for each synthesis step in the GMES scenario

5 Conclusion and Future Work

Enabling the dynamic composition of software components and solving their potential architectural mismatches is a complex challenge that can only be solved by appropriately combining different techniques and perspectives. In this paper, we consider both the software architecture and the middleware perspectives and propose an approach that brings together and enhances the solutions that seek to solve architectural mismatches from these perspectives. Our core contribution stems for the principled and rigorous approach to reason about components interaction using layered connectors, which formally describe components interaction at both the application and middleware layers. In addition, the systematic approach for synthesising layered connectors lays firm foundations for supporting dynamic composition in an increasingly heterogeneous world. The main idea is to first extract the data translations using knowledge about the application domain and to synthesise the appropriate controller that enables the components to interact successfully, then to refine this controller by taking into account the characteristics of the middleware solutions underneath.

As part of our future work, we would like to study the impact of errors or incompleteness in the specifications of the components or the domain ontology in the synthesis of layered connectors. This is even more relevant when the specifications are inferred using learning techniques. Therefore, we have to keep monitoring the components and their environment to detect changes and update the connectors accordingly. In this context, the incremental re-synthesis of layered connectors would allow us to respond efficiently to changes in the individual components or in the ontology. Another direction is to consider the security aspect, both on how enabling composition may induce unanticipated threats, but also how the increased ability to compose components dynamically may help securing software systems by rapidly reacting to newly discovered threats.

Acknowledgments. We acknowledge ERC Advanced Grant no. 291652 (ASAP).

References

1. Allen, R., Garlan, D.: Formalizing architectural connection. In: Proc. of ICSE (1994)
2. Bencomo, N., Bennaceur, A., Grace, P., Blair, G., Issarny, V.: The role of models@run.time in supporting on-the-fly interoperability. Computing (2013)

3. Bennaceur, A.: Dynamic Synthesis of Mediators in Ubiquitous Environments. Ph.D. thesis, Université Paris VI (2013), http://hal.inria.fr/tel-00849402/en
4. Bennaceur, A., Chilton, C., Isberner, M., , Jonsson, B.: Automated mediator synthesis: Combining Behavioural and Ontological Reasoning. In: Proc. of SEFM (2013)
5. Birrell, A., Nelson, B.J.: Implementing remote procedure calls. ACM Trans. Computing System (1984)
6. Bromberg, Y.D., Grace, P., Réveillère, L., Blair, G.S.: Bridging the interoperability gap: Overcoming combined application and middleware heterogeneity. In: Proc. Middleware (2011)
7. Ceriotti, M., Murphy, A.L., Picco, G.P.: Data sharing vs. message passing: synergy or incompatibility?: an implementation-driven case study. In: Proc. of SAC (2008)
8. Connect Consortium: Connect Deliverable D6.4: Assessment report: Experimenting with CONNECT in Systems of Systems, and Mobile Environments. FET IP Connect EU project. (2012), http://hal.inria.fr/hal-00793920
9. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch or why it's hard to build systems out of existing parts. In: Proc. of ICSE (1995)
10. Georgantas, N., Bouloukakis, G., Beauche, S., Issarny, V.: Service-oriented distributed applications in the future internet. In: Proc. ESOCC (2013)
11. Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge Acquisition (1993)
12. Inverardi, P., Tivoli, M.: Automatic synthesis of modular connectors via composition of protocol mediation patterns. In: Proc. of ICSE (2013)
13. Issarny, V., Bennaceur, A.: Composing distributed systems: Overcoming the interoperability challenge. In: HATS-FMCO International School. Springer Verlag (2012)
14. Issarny, V., Bennaceur, A., Bromberg, Y.D.: Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In: SFM-11 International School. Springer Verlag (2011)
15. Issarny, V., Kloukinas, C., Zarras, A.: Systematic aid for developing middleware architectures. Commun. ACM (2002)
16. Magee, J., Kramer, J.: Concurrency: State models and Java programs. Wiley (2006)
17. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of service protocols using process algebra and on-the-fly reduction techniques. IEEE Trans. Software Eng. (2012)
18. McIlraith, S.A., Son, T.C., Zeng, H.: Semantic web services. IEEE Intelligent Systems (2001)
19. Medvidovic, N., Dashofy, E., Taylor, R.: The role of middleware in architecture-based software development. Int. Journal of Soft. Eng. and Knowledge Eng. (2003)
20. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: Proc. of ICSE (2000)
21. Nitto, E.D., Rosenblum, D.S.: Exploiting adls to specify architectural styles induced by middleware infrastructures. In: Proc. of ICSE (1999)
22. Shaw, M.: Procedure calls are the assembly language of software interconnection. In: Proc. of ICSE Workshop on Studies of Software Design (1993)
23. Shvaiko, P., Euzenat, J.: A survey of schema-based matching approaches. J. Data Semantics IV (2005)
24. Spitznagel, B., Garlan, D.: A compositional formalization of connector wrappers. In: Proc. of ICSE (2003)
25. Tanenbaum, A., Van Steen, M.: Distributed systems: principles and paradigms. Prentice Hall (2006)
26. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM TOPLAS (1997)